

Towards Methods for Discovering Universal Turing Machines (or How Universal Unicorns can be Discovered, not Created)

James Harland¹

¹ School of Computer Science and Information Technology
RMIT University
GPO Box 2476, Melbourne, 3001
Email: james.harland@rmit.edu.au

Abstract

Universal Turing machines are a well-known concept in computer science. Most often, universal Turing machines are constructed by humans, and designed with particular features in mind. This is especially true of recent efforts to find small universal Turing machines, as these are often the product of sophisticated human reasoning. In this paper we take a different approach, in that we investigate how we can search through a number of Turing machines and recognise universal ones. This means that we have to examine very carefully the concepts involved, including the notion of what it means for a Turing machine to be universal, and what implications there are for the way that Turing machines are coded as input strings.

1 Introduction

Universal Turing machines (Turing 1936) are a well-known concept in computer science, and are most commonly used as a mathematical model of computation. Often, especially in textbooks, the intuitive, informal idea of a universal machine is all that is addressed (and for that matter, all that is necessary), in order to be able to prove undecidability results. With the benefit of 70-odd years of hindsight (and around 60 years of general purpose computing), the idea itself does not seem particularly difficult to grasp, and is often presented as an intuitively natural step from previous work.

Since Turing's original work, and particularly in recent times, there has been ongoing interest in finding small universal Turing machines (Minsky 1962, Watanabe 1972, Rogozhin 1996, Neary and Wood 2007). One particularly well-known effort is that of Minsky (Minsky 1962), who showed the existence of a 7-state 4-symbol machine that was universal. However, as Minsky himself pointed out, it is not at all obvious how one would 'program' this universal machine, nor indeed whether it is possible to identify this machine as universal just by inspecting it. Since then, a number of existence and non-existence results have been shown. These include the results that there are no universal Turing machines with 1 state, nor with 2 states and 2 symbols (Minsky 1962), nor 3 states with 2 symbols (Pavlotskaya 1978). There are also some results on more sophisticated measures of simplicity that merely the number of states (Calude

2008). Woods and Neary (Woods and Neary 2009) have given an excellent survey of the known results.

The notion of universal computation has become widespread within the culture of computer science¹, and the idea is often only discussed informally. This is usually done in the context of introducing the concept of universality in order to prove some undecidability results, such as the undecidability of the halting problem for Turing machines. However, the formal notion of universality is more difficult to find in the literature, and perhaps surprisingly, there is more than one precise notion. Essentially this difference is whether a universal machine has to precisely simulate the entire computation of the original machine (Fischer 1965), or whether it is sufficient to simulate the calculation of the output of the original machine from its input (Herman 1968). In the latter case, the universal machine does not need to simulate every step of the original computation, but can skip some steps, provided that the final result is the same. This difference can be thought of as a reflection of the differences between intuitive notions of computation, i.e. whether a computation computes an output, or carries out a process.

A different perspective on the problem of finding universal machines was given by Wolfram (Wolfram 2002), who performed an automated search for universal Turing machines (and for universality amongst other computational models such as cellular automata). This involved generating a large number of Turing machines and applying some criterion for evaluating whether or not it was universal. In Wolfram's case, this criterion was whether or not the output of the machine was 'inherently complex' or not, i.e. it was not a straightforward, regular pattern, but contained some seemingly random elements.

Whilst this approach is commendable, and is similar in spirit to automated searches to find values for the *busy beaver* function (Rado 1963, Lin and Rado 1964, Brady 1983, Marxen and Buntrock 1990), the criterion used essentially relies on human judgement, and was not specifically defined, which makes the results difficult to analyse (and reproduce). However, it does appear to be the first systematic attempt to search for universal machines, rather than to construct them.

A further issue raised by Wolfram's search is that his universal machines necessarily do not terminate; in order to simulate a terminating computation, the final configuration of the terminating computation is repeated indefinitely in the simulation. This adds an extra dimension to the discussion of universality, in that it needs to be established in advance whether this is a reasonable property, or whether it is appropriate to insist that a terminating computation can only be simulated by a terminating computation on

Copyright ©2011, Australian Computer Society, Inc. This paper appeared at the 17th Computing: The Australasian Theory Symposium (CATS 2011), Perth, Australia, January 2011. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 119, Alex Potanin and Taso Viglas, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

¹Dare one say that the concept of universal machines has become universal? :-)

the universal machine.

In this paper, we discuss how to perform a systematic search for universal Turing machines, but with more explicit criteria. This means we need to examine the notion of universal machine rather closely. In particular, we need to be precise about exactly what it means to be universal, and what precise criteria can be applied in an automated search. However, as we shall see, this is quite a large undertaking; not only are there various issues in the definition of universality to be dealt with, including the definitions of appropriate encoding functions, there are also a number of difficulties in defining a universality criterion which is sufficiently precise to be able to completely determine the universality of a given machine. In particular, it seems likely that the best that we can hope for is a *pseudo-universality* test, similar in spirit to pseudo-primality tests, which can be used to definitively determine that a given number is not prime, but can only establish primality with a level of uncertainty.

The main contribution of this paper is to set up a framework in which to investigate universality properties of classes of machines. This paper is organised as follows. In Section 2, we provide a detailed discussion of the issues involved in the definition of universality, and in Section 3 we give our explicit choices of Turing machine and related formal issues. In Section 4, we discuss how to generate the classes of candidate machines, and in Section 5 we discuss issues relating to the encoding of machines. In Section 6 we provide a summary of this discussion in terms of a list of issues to be resolved, and in Section 7 we discuss various outstanding issues. Finally in Section 8 we present our conclusions and areas of further work.

2 Discovering Universal Machines

2.1 Definitions of Universality

Alan Turing introduced the formal model of computation which is now known as Turing machines (Turing 1936), and showed that it is possible to construct a universal Turing machine. This machine takes another Turing machine as input and is able to simulate the computation of the original machine with a suitable input to the universal machine. Whilst the main purpose for Turing's introduction of the universal machine was to show the halting problem for Turing machines is undecidable, there has been interest since then in finding small universal Turing machines. This has generally been done by construction, i.e. by finding particular machines which are increasingly small. As mentioned above, there are various results known about small universal Turing machines, and there is ongoing work aimed at 'closing the gap' between the largest known classes of machines in which there are no universal machines, and the smallest known universal Turing machines (Woods and Neary 2009). Whilst the ongoing search for small universal Turing machines is a fascinating and ongoing research topic, it is not the focus of this paper. Here we are concerned with how we can discover universal machines within a given set of machines.

In order to perform a search along the lines of Wolfram's, there are a number of issues that need to be addressed. A first observation is that this search will be similar to searches undertaken to solve the busy beaver problem (Rado 1963, Lin and Rado 1964, Brady 1983, Marxen and Buntrock 1990, Harland 2006, 2007). The busy beaver problem is to find the largest number of 1's that can be printed by a terminating Turing machine with no more than n states on the blank input (often referred to as the *productivity* of the machine). Hence the search for a busy

beaver (and related concepts such as the *placid platypus* (Harland 2006, Bátfai 2009)) involves generating all machines in the class, evaluating whether or not each machine terminates on the blank input², and finding the maximum productivity of the terminating machines. It is well-known that the busy beaver function is not computable (Rado 1963); however, as there are only a finite³ number of machines of a given size, it is possible to compute the value of the busy beaver function for any given input value.

To apply a similar process to finding universal machines, we need to be able to determine whether a given machine is universal or not. This is a different perspective from the way that universal machines are usually presented in textbooks, as universal machines are normally produced by construction, together with an appropriate encoding function. However, in order to search for universal machines amongst series of Turing machines, it is necessary to develop a particular criterion for evaluating the universality or otherwise of a given machine.

One issue that arises in the contemplation of such a criterion is that not only is it comparatively rare to find a formal definition of a universal Turing machine, but that there are *at least two such definitions*. One may be termed *result-oriented*, in that given a Turing machine M with an input w , a universal machine U need only "agree" with the result of the computation of M on w (Neary and Wood 2007, Priesse 1979, Herman 1968, Davis 1956, 1957). In other words, a machine U is universal if for any machine M and input w to M , there is an input w' to U such that the result of the computation of U on w' is an encoding of the output of M on w . In particular, it is not necessary for each step of the computation of U on w' to 'mimic' every step of the computation of M on w . Note that w' is anything but arbitrary; typically w' is constructed in a very precise manner from M and w . The other definition, which may be termed *process-oriented*, insists that each step of the computation of M on w is in fact mimicked by the computation of U on w' , so that for each configuration in the computation of M on w , there is a corresponding unique configuration in the computation of U on w' (Fischer 1965). It is not hard to see that a machine which is universal in the process-oriented sense must also be universal in the result-oriented sense, as in the former case, we certainly require that the final configuration in the computation of M on w corresponds to the final configuration of the computation of U on w' .

The variance between these definitions is perhaps not all that surprising, in that in order to define a universal machine, it is necessary to give a precise definition of computation, and at least in an intuitive sense, computation can be thought of as either a process which takes a given input and computes a specific output, or as a process which follows a particular series of steps, and which may not necessarily terminate (and in fact not terminate by design, rather than as a consequence of the existence of undecidable problems). This same variance of opinion is at the heart of the two different definitions of universality (although it must be said that the result-oriented approach seems to have achieved a recent consensus). This view is perhaps best summarised in the quotation below.

"... the purpose of the simulation is to reproduce the input-output relation of the simulated machine."

– Gabor Herman, (Herman 1968), page 813.

²Note that as there are only a finite number of machines involved, this is a decidable problem; see the discussion on the finite decision anomaly in §7.

³but hyperfactorial, i.e. $O(n^n)$

However, if we think of computation as a process, which includes the possibility of non-termination, then it seems more natural to adopt the process-oriented definition. In particular, the result-oriented definition treats all non-terminating computations as the same (and uninteresting), which may not seem intuitive for software systems such as web servers, or operating systems, or other software which does not fit neatly into what one may call an algorithmic view of computation. In addition, the result-oriented approach has the effect of ignoring differences between computations which produce the same output. For example, if we only consider the input-output behaviour of programs, then all sorting algorithms are effectively considered to be the same. To be fair, the result-oriented view isn't always this extreme, in that it is not unreasonable to allow a universal machine to 'skip' a few steps in the original computation. However, it seems difficult to define precisely what is meant by such harmless skipping without also allowing the entire computation of M on w to be simulated by a single step of U on w' . Of course it is unlikely that such a universal machine exists (i.e. one that simulates the computation of M on input w with a single step); the point here is that before we can begin to search for such machines, we need to consider carefully what exactly we are looking for. Also, one lesson that can be learned from the search for busy beavers is that human construction of machines is generally no match for those that can be found by systematic searches, particularly when there are restrictions on the size of the machines that may be used. Hence it would seem best to avoid as much as possible relying on assumptions that certain properties are 'unlikely'. To exaggerate to make the point, the productivities of the 6-state 2-symbol busy beaver candidates seemed 'unlikely' until they were found.⁴

2.2 Termination

Another issue that needs to be addressed is whether or not a universal machine must mimic the termination behaviour of the original machine. More particularly, must the simulation of a terminating computation terminate? (and, for that matter, must the simulation of a non-terminating computation also not terminate?). As mentioned above, it is not possible for Wolfram's machines to terminate, as there is no halt transition. This raises the issue of whether a universal machine must be of the same 'type' as the machines it simulates. There is an intuitive sense in which a universal machine must be able to simulate its own computations (as otherwise it is in some sense not universal, as there is at least one machine whose computations it doesn't simulate). However, this can also be viewed as asking over what class of machines the universal machine is expected to operate. In terms of simulating all computations for all machines in a given class, it is not obvious that the only class of interest is all possible machines. For example, it may be of interest to consider only machines of a certain computational complexity, or which perform a specific set of algorithms (such as arithmetic operations, or sorting algorithms). It is certainly natural to require that a universal machine be able to simulate all possible machines, including itself, but again this may be making assumptions which, while they appear natural, may not be justified. Another way to put this is that universality, in its intuitive sense, is one extreme; it may be that a given machine will simulate some computations and not others, and so a universal machine may be considered as one kind of *meta-machine* (i.e. a machine which takes

another machine as input). It is intriguing to consider other possibilities for interesting meta-machines, but in this paper we will consider only universal machines (whichever precise definition is used). We will refer to universal machines whose termination behaviour precisely matches that of every machine which is input to it as *faithful*; in other words, a faithful universal machine must terminate when the machine it simulates terminates, and not terminate when the machine it is simulating does not terminate.

2.3 Encoding Machines

Another issue that needs to be considered is how the machine to be simulated is encoded as an input to the universal machine. As noted by Herman (Herman 1968) (see below), it is important that the coding function serve *only* to translate the machine into a form that can be read by another machine.

"This is to stop the encoding and decoding of algorithm to carry out the real computational work"

– Gabor Herman, (Herman 1968), page 813.

Similar concerns are discussed by Colmerauer (Colmerauer 2004). This occurs in his description of his development of a very sophisticated universal Turing machine, which is notable not only for being explicitly defined and well-documented, but also in the amount of effort put into its design, making it arguably the most sophisticated universal Turing machine known. Colmerauer's concerns are not about making universal Turing machines small, but about providing an appropriate metric for measuring the overhead involved in simulating one machine by another. Specifically, Colmerauer aims to construct a universal machine which minimises this particular metric. The details of this machine are beyond the scope of this paper, but much of Colmerauer's analysis is relevant, and so we will discuss its important aspects.

Colmerauer also discusses the encoding function in some detail, and is in particular concerned about coding functions that may 'cheat', such as by recognising particular machines, and encoding these differently from other machines. It seems natural that one way to avoid any such problems is to place some particular requirements on coding functions. In particular, it seems reasonable to require that the function itself must be defined *homomorphically*, i.e. in terms of its effects on the states, symbols and other elements of the machine, and let these definitions determine the overall encoding of the machine. This makes it impossible for the encoding to recognise particular machines, or to perform 'the real computational work', as all it is doing is to translate the nuts-and-bolts of one machine into the input language of another.

Another important observation that Colmerauer makes is that the encoding itself is bound up in the definition of universality. In particular, the design process for building a universal machine usually commences with defining an appropriate encoding, which is then used throughout the universal machine. It is difficult to imagine a machine that has been defined with one encoding in mind being sufficiently robust to also be universal for a different encoding, no matter how similar. Put another way, we can hardly expect a machine to retain the seemingly brittle and scarce property of universality if we were to change the encoding on which it depends, but not the machine itself. For these reasons Colmerauer refers to a universal machine and its encoding function as a *universal pair*.

⁴The best known such productivity is currently $10^{18,276}$ (!).

It is also notable that Colmerauer defines what we may call a particular *architecture* for the way in which the universal machine works. In a nutshell, given a machine M and an input w , which is assumed to be using the same symbols as the universal machine U , the input to U which is used to simulate the computation of M on w is $code(M)w$ where $code()$ is the encoding function. The important point to note is that we have made a particular assumption about the way in which the universal machine will read and organise its input (and in particular that the initial state of U will be pointing to the first symbol to the left of the string $code(M)w$). When universal machines are constructed by hand, this is entirely natural; when we are searching for universal machines amongst a number of machines, it is not so obvious that it is appropriate to be so specific about how the machine is organised. For example, what if there is a machine which expects its input in the form $w\ code(M)$? This may seem to be a rather vacuous objection, in that clearly the universal machine needs to be able to access the information about M and w somehow. However, we again need to be careful about making unwarranted assumptions.

This may indicate that an appropriate response is to not specify exactly what the input to the universal machine should be. In this case, a machine U would be universal if for each machine M and input w there is a w' such that M on w is simulated by the computation of U on w' . However, this may allow some trivial machines to be universal, as if w' is just the output of M on w (assuming M terminates on w), and U is the machine which halts immediately, then clearly the output of U on w' is the same as the output of M on w . Note also that this is only an issue for the result-oriented approach; in the process-oriented approach, even if w' is an encoding of the result of M on w , we are still required to perform a computation of U on w' for which each state of the computation of M on w is represented by a unique configuration in the computation of U on w' .

Hence it seems reasonable to require that the input w' to U be an encoding of M and w . The main issue arises when the test for universality fails (which we expect will be most of the time). In particular, it is not obvious whether it is reasonable to try other permutations of the input or not (such as $w\ code(M)$). It should also be noted that it is possible to have variations on universality such as weak universality (Woods and Neary 2009), in which an infinite number of copies of a particular string are kept on the tape. Whilst we do not pursue this and similar options here, it is worth noting that such variations can be thought of as one aspect of what we have termed the architecture of the universal machine.

2.4 Pseudo-universality

Perhaps the most interesting aspect of Colmerauer's architecture is the following reasoning. Colmerauer defines a universal machine U as one that for any M and w , U on $code(M)w$ simulates M on w for some appropriate function $code()$. Now as M can be any Turing machine, M could in fact be U , and so if $M = U$, we must have that for any w , U on $code(U)w$ simulates U on w , and so U on $code(U)code(U)w$ simulates U on $code(U)w$ which simulates U on w . By extending this argument, we have that U on $code^k(U)w$ simulates U on w for all $k \geq 1$.

Colmerauer then uses this property to define his metric. In our case, the interesting aspect of this observation is that we can use this property together with an analysis of busy beaver candidates to arrive at a "pseudo-universality" test. As U on $code^k(U)w$

simulates U on w for all $k \geq 1$, then when w is the blank tape (written here as \sqcup), then we have U on $code^k(U)\sqcup$ simulates U on \sqcup for all $k \geq 1$. In other words, for a universal machine U , U on $code^k(U)\sqcup$ simulates the computation of U on the blank input, which is exactly what is tested for in the busy beaver analysis. Hence we arrive at the following definition.

Definition 1 A deterministic Turing machine M is pseudo-universal with respect to $code()$ if M on $code^k(M)\sqcup$ simulates M on \sqcup for all $k \geq 1$.

Clearly any universal machine will be pseudo-universal, but not necessarily vice-versa. This is intentionally similar to pseudo-primality tests, which are based on necessary (but not sufficient) properties of primes. In this case we need only find some j for which M on $code^j(M)\sqcup$ does not simulate M on \sqcup to show that M is not pseudo-universal. As we expect universal machines to be rare, and any machine universal with respect to $code()$ will be pseudo-universal, we expect a relative small number of machines to pass the pseudo-universality test. In this way, Colmerauer's property leads to one notion of a criterion for testing for universality (or more specifically, for non-universality). It seems natural to measure the quality of the pseudo-universality test by measuring the number of machines that are pseudo-universal but not universal (and clearly the fewer there are, the better the test is).

3 Formal definitions

Before examining some of the issues discussed above in more detail, we give some formal definitions, including a precise specification of the Turing machines that we will be using.

We use the following definition of a Turing machine (Sudkamp 2005).

Definition 2 (Sudkamp) A Turing machine is a quadruple $(Q \cup \{h\}, \Gamma, \delta, q_0)$ where

- Q is a finite set of states (and $h \notin Q$)
- h is a distinguished state called a halting state
- Γ is the (finite) tape alphabet
- δ is a partial function from $Q \times \Gamma$ to $Q \cup \{h\} \times \Gamma \times \{l, r\}$ called the transition function
- $q_0 \in Q$ is a distinguished state called the start state

In this paper, Q is $\{a, b, c, d, \dots\}$ unless otherwise specified, where a is the initial state (i.e. q_0 is labelled a).

Note that this is the so-called quintuple transition variation of Turing machines, in that a transition must specify for a given input state and input character, a new state, an output character and a direction for the tape in which to move. Hence a transition can be specified by a quintuple of the form

(State, Input, Output, Direction, NewState)

For this reason we will often refer to a transition in a given machine as $t(S, I, O, D, NS)$.

Some varieties of Turing machines only allow a transition to write a new character on the tape or to move, and not both; for such machines, clearly only a tuple of 4 elements is required. For our purposes, the important thing to note is that given some notational convention for identifying the start state and

halting state (here we denote the start state as a and the halting state as h), a Turing machine can be characterised by the tuples which make up the definition of δ .

Note also that there are no transitions for state h , and that as δ is a partial function, there is at most one transition for a given pair of a state and a symbol in the tape alphabet.

Definition 3 Let (S, I, O, D, NS) be a transition in a Turing machine M . We say this is a halting transition if $NS = h$.

We call a Turing machine M

- **normal** if there is exactly one halting transition in M .
- **exhaustive** if δ is a total function from $Q \times \Gamma$ to $Q \cup \{h\} \times \Gamma \times \{l, r\}$, i.e. that $\forall q \in Q \forall \gamma \in \Gamma \exists q' \in Q \cup \{h\}, \gamma' \in \Gamma$ and $D \in \{l, r\}$ such that $\delta(q, \gamma) = \langle q', \gamma', D \rangle$.

Note also that machines which are exhaustive but not normal are either guaranteed not to terminate (as there is no transition into the halting state and every combination of state and input symbol has a transition defined for it), or have multiple halting transitions, of which at most only one can ever be used, making the other halting transitions spurious. In our terminology, Wolfram's machines are exhaustive, but not normal. We in general will be interested in normal machines, but not necessarily exhaustive ones.

We denote by an n -state Turing machine one in which $|Q| = n$. In other words, an n -state Turing machine has n "real" states and a halt state.

As we will often be discussing universality, which necessarily involves taking one Turing machine as the input machine and another as the (potentially) universal machine, we will use the term *input machine* to refer to a machine whose computation is to be simulated and *candidate machine* to refer to the potentially universal machine. In other words, when discussing whether the computation of M_1 on w is simulated by M_2 on $\text{code}(M_1)w$ (or whatever input is appropriate), we refer to M_1 as the input machine and M_2 as the candidate machine.

Following the busy beaver linguistic precedent⁵, we will call universal machines *universal unicorns*.

4 Searching Classes of Machines

When generating classes of machines to be tested, it is tempting to directly re-use the classes already generated for the busy beaver searches. However, these classes are understandably optimised for searching for busy beaver machines. As the number of machines of a given size is hyperfactorial (i.e. $O(n^n)$), it is clearly important to minimise the number of machines that need to be searched.

One way in which this is done for busy beaver machines is to make strong use of the fact that the initial input is blank. Given that the tape is symmetric, the direction moved by the initial transition may be arbitrarily chosen, and is usually chosen as r .⁶

Let us assume that there are only two tape symbols, including blank, which we denote as $\{0, 1\}$ where 0 is the blank (this convention is standard in the busy

beaver literature). If we denote the initial transition as $t(a, 0, O, r, NS)$ (note that I must be 0 for the first transition used, as the input is blank), then if $NS = a$, then the machine will loop infinitely to the right, no matter what the value of O is, as the computation is started on a blank tape. Hence it is sensible to insist that $NS = b$,⁷ and so we have the first transition as $t(a, 0, O, r, b)$. Now either $O = 0$ or $O = 1$. If $O = 0$, then note that the first step of the machine does not change the tape at all. Hence we could replace this machine with the one that results from swapping the state a with the first state S for which we have $t(S, 0, 1, D, NS)$, i.e. the first state for which $I = 0$ and $O = 1$. If there is no such state, then the machine cannot print any 1's, and hence is not of interest. The new machine is then one that prints a 1 as a result of the first transition, and which otherwise has the same productivity as the original. This means that for busy beaver machines, we can always assume the first transition is $t(a, 0, 1, r, b)$.

Another consideration is that busy beaver machines are assumed to be normal and exhaustive. Being normal ensures that there is a way for the machine to terminate, which is obviously vital for busy beaver machines. Being exhaustive is a property that helps with maximising productivity. A terminating machine that is not exhaustive is in some sense wasteful. For example, consider a machine which contains the transition $t(c, 0, 1, r, h)$ but has no transitions for state b with input 1 and state c with input 1 (which may be considered as having three halting transitions). By halting when the machine first encounters 0 in state c , the machine is in some sense "missing out" when compared to a machine which defines a new transition for 0 in state c , and then continues, in the knowledge that there are still at least two possibilities for the halt transition (i.e. 1 in state c and 1 in state b). In other words, if there are less than $n \times m$ transitions, say k , then there is another machine with the same k transitions and with another added which will have higher productivity⁸. Hence, whilst there is an explicit halt transition in these machines, the generation of the machines is carefully constrained to ensure that each machine contains exactly $n \times m$ transitions. Effectively this means that there are 'only' $n \times m - 2$ transitions that need to be generated, as the first one is fixed, and when there is only one transition left to be specified, not only must this be the halt transition, it can always be assumed to be of the form $t(S_1, I_1, 1, r, h)$, where h is the halt state (i.e. there are no transitions from state h). The direction is again arbitrary (as the machine halts after this transition, the position of the tape pointer is irrelevant), and as it is desired to maximise the number of non-blank symbols printed by the machine, it is always sensible for the output to be 1.

A third issue is the technique that is generally used to generate machines known as *tree normal form* (TNF) (Lin and Rado 1964). This may be thought of as exploiting the property that machines are normal and exhaustive, in that machines are generated by computing with partially defined machines and adding transitions when a combination of state and input is encountered which is not yet defined in the machine. Initially the machine consists of just the transition $t(a, 0, 1, r, b)$. Executing the partial machine on a blank tape gives us the configuration $1\{b\}0$ (where we use the convention that the tape pointer is

⁵Also known as the law of *alliterative adjectives*.

⁶Hence any machine in this class has a *sinister sibling* machine in which the direction moved by each transition, including the first, is reversed. The original machine and its sinister sibling will clearly behave identically apart from the direction of motion of the tape head.

⁷Note that when generating such machines, we use the methodology that the first state that is not a is b , the first state that is neither a nor b is c , and so forth. Otherwise, we run into unnecessary combinatorial explosions.

⁸Whilst this informal argument seems reasonable, it should be said that to the best of the author's knowledge, this argument has not been formalised.

pointing to the cell immediately to the right of $\{b\}$). As we have no transition defined for the state b with input 0, we choose values for O, D and NS , and add the transition $t(b, 0, O, D, NS)$ to the machine. We then update the configuration according to the newly generated transition and continue. This process continues until we have only two possible transitions left. Once we allocate one of these, we then know that the final remaining combination of state and input (say S_h and I_h) must be for the halting transition, and so we complete the machine by adding the transition $t(S_h, I_h, 1, r, h)$.

When searching for universal unicorns, the assumptions on which these simplifications are based do not apply. Firstly, the input is not always going to be blank, and so the first transition will not always be $t(a, 0, 1, r, b)$. Secondly, it is not obvious that a universal machine will be exhaustive, i.e. contain $n \times m$ transitions⁹, and so we will need to allow the generation of machines which have between 2 and $n \times m$ transitions in them. Also, the halt transition cannot be assumed to be of the form $t(S_1, I_1, 1, r, h)$, as the final state of the machine may require the tape pointer to move either left or right or to output an arbitrary symbol in order to terminate with an appropriately encoded state of the machine. Finally the TNF method, which works very well for busy beaver machines, cannot be used here, both because we cannot assume that the input is blank (and hence we do not know which transition will be used first) and also because we do not assume that the machine must be exhaustive. The latter objection could be overcome by allowing the halting transitions to be chosen ‘freely’, rather than always being the last to be chosen. However, the former objection is more fundamental, as the TNF technique is very much predicated on the knowledge of a single fixed input, which will clearly not apply to a universal machine.

Hence whilst the results of the search for busy beavers will provide some useful data against which to compare results, the search space is smaller, and so we will need to perform a larger search for universal unicorns. In particular, we will need to ‘freely’ generate machines, rather than in the sequential manner implied by the TNF process. For example, the TNF process for 4 states and 2 symbols generates 527,590 machines to be tested. In the universal case, we will require 8 transitions, each of which will have 2 (for O) \times 2 (for D) \times 5 (for NS) = 20 possibilities. Note that there are 5 possibilities for NS , as it is possible for any given transition to be the halting transition. This means that there are $20^8 = 2^8 \times 10^8 = 25,600,000,000$ machines to be searched. It may be possible to reduce this number by some clever analysis (such as not allowing the first transition used to be the halting transition), but in any case it is clear that this will involve a substantially larger search than the busy beaver case.

5 Encoding Machines

A key issue, as we have seen, is the precise properties of the encoding function. In this section we discuss various issues related to this function.

5.1 Coding Functions

A natural place to start looking for such functions is to see what functions have been used in the literature. In particular, there are a large number of textbooks which discuss universal Turing machines,

⁹Although this is probably likely, especially for small machines.

and whilst most such treatments give only an informal description of a universal Turing machine, most such accounts explicitly define a coding function.

A good example is one given by Sudkamp (Sudkamp 2005) (3rd edition, pp. 327-328). Sudkamp informally describes a 3-tape universal Turing machine, in which Tape 3 contains the simulated tape of M and hence has w on it at the beginning of computation, Tape 2 has the current state and Tape 1 has the encoded version of M . This machine is described rather than explicitly given. However, our main interest here is the coding function, which is as follows (both the original machine and universal machine use the input alphabet $\{0, 1\}$ and the tape alphabet $\{B, 0, 1\}$). We will define the encoding in terms of three functions $inputs()$, $states()$ and $ops()$, as below.

$$\begin{aligned} inputs(0) &= 1 & states(q_0) &= 1 \\ inputs(1) &= 11 & states(q_1) &= 11 \\ inputs(B) &= 111 & \dots & \\ & & states(q_n) &= 1^{n+1} \\ ops(L) &= 1 & ops(R) &= 11 \end{aligned}$$

A transition $t(S, I, O, D, NS)$ is encoded as the concatenation of the encoding of each of its elements, with a 0 inserted between each element. Hence a transition is encoded as below.

$$states(S)0inputs(I)0states(NS)0inputs(O)0ops(D)$$

0 is used to separate components of a transition, 00 is used to separate transitions, and 000 is used to indicate the start and end of the encoded machine.

Hence if $M = [t_1, \dots, t_n]$, then we have (writing $tr(t_i)$ for the encoding above):

$$code(M) = 000tr(t_1)00tr(t_2)00 \dots 00tr(t_n)000$$

Note that this may be classified as a unary code, in that a variable number of 1's is used to specify which of the different states, inputs and directions is meant, and the 0 is used only as a separator. It is interesting to note that many textbooks have either a similar code (i.e. some variation of a unary code), or a code which involves a large number of symbols, which is designed to simplify the construction of the universal machine. Given that we will be testing a given machine for universality, we are not in a position to choose the number of symbols that may be used, and so we will have to allow for a number of possible codes.

5.2 Unary Codes

With this example code in mind, it is not hard to specify some general conditions for unary codes to ensure some minimal ‘sensible’ properties.

Let S be the set of states used in all machines in \mathcal{M} (so that for any $M \in \mathcal{M}$, the states of $M \subseteq S$). Let $names(S)$ be an ordered sequence of names for elements of S ; for convenience, we will assume that $names(S)$ is the sequence s_1, s_2, s_3, \dots . If S is finite then clearly so is $names(S)$. Hence for any $s \in S$, there is an i such that $name(s) = s_i$. We denote by $n(s)$ the sequence s_1, s_2, \dots, s_i . Let Ops be the set of all operations used in all machines in M . Usually $Ops = \{l, r\}$.

A coding function is then defined in terms of three components: a mapping on S , a mapping on Δ and a mapping on Ops , denoted as $states$, $inputs$ and ops respectively. We also assume that the coding function is generated homomorphically from its definition on individual transitions. This eliminates some of Colmerauer’s ‘cheat’ cases, in that

there can be no definitions like “if the machine is U then the empty string else ...”. This means that if M has the transitions t_1, \dots, t_n , then $\text{code}(M) = \text{code}(t_1)\text{code}(t_2) \dots \text{code}(t_n)$.

To avoid ‘lazy’ codes, which for example encode the input symbol 1 as an entire machine, we insist on the following properties:

- For all $s \in S$, $|\text{states}(s)| \leq |n(s)| + 1$
- For all $i \in \Delta$, $|\text{inputs}(i)| \leq |\Delta| + 1$
- For all $o \in \text{Ops}$, $|\text{ops}(o)| \leq |\text{Ops}| + 1$

This means that codes can be at worst unary, and generally better than that. For example, if our “target” universal machine has tape alphabet $\Sigma = \{x, y\}$, then a maximally “worst” permissible coding would be one that maps the states $\{s_1, s_2, \dots, s_n\}$ to $\{yx, yyx, yyyx, \dots, y^n x\}$.

Given such a unary code, it is not hard to see that we do not need to consider the particular way in which the symbols are used. If the language has at least three symbols (including the blank symbol), which we will denote as $\{B, 0, 1\}$, then it is not hard to see that it does not matter whether we choose 0 as the ‘separator’ and 1 as the ‘counter’, as above, or 1 as the separator and 0 as the counter. In particular, if M is a Turing machine and we denote the former alternative as $\text{code}()$, then for any machine M which is universal with respect to $\text{code}()$, then there is a machine M^- which is universal with respect to $\text{code}^-()$, where $\text{code}^-()$ is the same as $\text{code}()$, except that the roles of 0 and 1 are interchanged, and M^- is M under the following transformation:

For each transition $t(S, I, O, D, NS)$ in M , we have a transition $t(S, \text{swap}(I), \text{swap}(O), D, NS)$ in M^- , where $\text{swap}(0) = 1$ and $\text{swap}(1) = 0$.

When the language contains only two symbols (including a blank), then the same transformation can be performed. In fact, if we denote the language (perhaps confusingly!) as $\{B, 1\}$, then it is arguably simpler to use 1 as the separator and B as the counter, as then the representation of a machine on an otherwise blank tape will be

111B1B1BB1BB1B11BB...111

Whichever version is preferred, we can identify a ‘canonical’ unary code in this way, knowing that if there is a universal machine for one version of the code, then there will be a universal machine for the other version. There is some potential for variance amongst the way that transitions are separated and whether the 000 at the start and end of the machine is strictly necessary. However, it is clear that one does not need more symbols used in this way than in Sudkamp’s code, and arguably less. In addition, we clearly require at least one ‘separator’ between elements of an encoded transition and at least one at the beginning and end of the machine. This means that any savings on such separators will be at most one symbol per transition plus 4 overall (as the 000 and 000 at the start and end of the machine could be just 0 and 0). As there are three different kinds of separator needed (between elements of a transition, between transitions, and at the start and end of a machine), there are four variations of this code that seem reasonable to try:

1. All separators are distinct (as above)
2. Two are the same and one is not
3. All separators are the same

Hence if we denote the separator between elements of a transition as Sep_1 , between transitions as Sep_2 , and at the start and end of a machine as Sep_3 , we have the following four cases:

1. $\text{Sep}_1 = 0, \text{Sep}_2 = 00, \text{Sep}_3 = 000$ (as in Sudkamp)
2. $\text{Sep}_1 = \text{Sep}_2 = 0, \text{Sep}_3 = 00$
3. $\text{Sep}_1 = 0, \text{Sep}_2 = \text{Sep}_3 = 00$
4. $\text{Sep}_1 = \text{Sep}_2 = \text{Sep}_3 = 0$

It is possible that there could be other codes, such as using $\text{Sep}_1 = 000000$, but we are assuming that this would not be used in preference to one of the shorter codes above. This means that we could reasonably limit the search for a universal Turing machine to one of the above four cases, and to the case when 0 is the separator symbol and 1 is the counter symbol.

5.3 Binary Codes

A natural generalisation of unary codes is to use binary ones. To represent an n -state m -symbol machine in an alphabet containing k symbols, we require the following:

Element	S	I	O	D	NS
Possible cases	n	m	m	2	$n + 1$

Note that $n + 1$ possibilities are needed for NS rather than n as we need to allow for the halting state. This means that each transition will require

$$\lceil \log_k n \rceil + \lceil \log_k m \rceil + \lceil \log_k m \rceil + \lceil \log_k 2 \rceil + \lceil \log_k (n + 1) \rceil$$

symbols to encode it, and so each machine requires up to

$$nm(\lceil \log_k n \rceil + \lceil \log_k (n + 1) \rceil) + 2\lceil \log_k m \rceil + 1$$

symbols (note that $\lceil \log_k 2 \rceil = 1$ as $k \geq 2$). We are assuming that there are no separators necessary here; either we add them as above, or we can assume that these are not necessary if n and m are known in advance. Knowing n and m in advance means that we can determine the exact number of symbols used to represent each of the elements of a transition, the length of each encoded transition and the maximum number of transitions, and hence no separators are needed.

Hence given k symbols, in principle one can look at codes of dimension $1, 2, \dots, k$. It is tempting to say that a code of dimension $k - 1$ is unlikely to be used if one of dimension k is possible; put a little more starkly, why would anyone use a unary code if a binary one could be used? Whilst this is a reasonable attitude for humans creating universal machines, when it comes to searching a given class for universality, it seems highly appropriate to allow for such ‘improbable’ codes. One lesson that can be drawn from the search for busy beavers is that humans are not very good at constructing machines with complex behaviours and a limited number of states, and so we should be very careful about what assumptions are made (and be very precise in stating exactly what they are).

Now if our candidate machine M is used as an input machine, then we can assume that $k = m$. This means that each machine will require at most

$$nm(\lceil \log_m n \rceil + \lceil \log_m (n + 1) \rceil) + 3$$

symbols, as $\log_m m = 1$. Now if we assume that $n < m$, and so $n + 1 \leq m$, this becomes $nm(1 + 1 + 3) = 5nm$.

This shows that machines which have more symbols than states can be encoded more compactly than machines with more states than symbols. This in itself is not a particularly deep observation; however, this may help to explain a rather mysterious property of the busy beaver machines, which is that the n -state 2-symbol machines seem to be a less complex class than the 2-state n -symbol machines. Consider the table below.

States	Symbols	Productivity	Steps
2	2	4	6
3	2	6	21
2	3	9	38
4	2	13	107
2	4	2,050	3,932,964
3	3	374,676,383	119,112,334,170,342,540
5	2	≥ 4098	$\geq 47,176,870$
2	5	$\geq 1.7 \times 10^{352}$	$\geq 1.9 \times 10^{704}$

This greater productivity may reflect the greater compactness of representation possible. More particularly for the search for universal machines, this suggests that we are more likely to find universal machines in the 2-state n -symbol class than the n -state 2-symbol class.

6 Process

So in order to perform an automated search for universality, we need to do the following:

1. *Determine the appropriate definition of input machines.* This will include whether or not the machine is deterministic, how many tapes, and tape heads it has, whether the tape (or tapes) is infinite in one direction only or both directions, and whether or not an explicit halting transition is required.
2. *Determine the appropriate definition of candidate machines.* Generally this would be the same class of machines as the input machines, but this method will allow for possibilities such as Wolfram's universal machines, which always do not terminate, and terminating machines are simulated by infinitely repeating the encoding of the final configuration of the terminating machine.
3. *Determine whether or not universal machines are required to be faithful.* This may often be implicit from the class of machines chosen in 2 above, but it seems appropriate to insist that an explicit statement be made.
4. *Determine the precise notion of universality;* in particular, it should be stated whether this notion is result-oriented or process-oriented.
5. *Determine the architecture.* This involves choosing the general strategy of how the input machine will be presented to the candidate machine, such as how many copies of the input machine are used, and where the copy or copies will be placed on the tape. We will assume that only one copy is used unless explicitly stated otherwise.
6. *Determine the encoding.* If there are k symbols in the language of the candidate machine, it seems that it will be necessary to try at least one i -ary code for each $1 \leq i \leq k$.

7. *Determine an appropriate criterion to be applied to the class of candidate machines.* This is likely to be a necessary but not sufficient test (like the pseudo-universality discussed above).

We refer to 1-6 above as the *universality context* in which the universality criterion is evaluated.

If we find a machine which passes the test in 7, then either we have found a universal machine, or at least a likely candidate for one, which may require further analysis. Machines which fail the test can be definitively stated to be not universal for this universality context, and for any reasonable criterion and universality context, it seems reasonable to expect that most machines will fail. However, in order to state that a machine is not universal for any universality context will clearly take significantly more work.

The above process will be used to design implementations which generate appropriate classes of machines and test them for universality. We have commenced work on such an implementation, but it is not yet complete.

It should be noted that such empirical searches will always have limitations; in this case, it would appear to be the variety of different universality contexts, and in particular, the variety of possible encoding functions. However, it is hoped that even some partial success in this way will help shed some light on the issue of how small universal machines can be expected to be.

7 Implementation

We are developing a prototype implementation which will initially use the following universality context:

1. normal exhaustive machines
2. normal exhaustive machines
3. faithful machines
4. process-oriented universality
5. Colmerauer's architecture
6. Sudkamp's encoding

We will use the pseudo-universality criterion of Definition 1 in this universality context.

We will first apply this universality criterion in this universality context to the class of 2-state 2-symbol machines. Whilst it is known that there are no universal machines in this class, this will give us some indication of how precise our criterion for pseudo-universality is. As we are clearly unable to directly test M on $code^k(M)$ for all $k \geq 1$, we will test all values of k up to some maximum value, such as 10.

The most difficult task will be to evaluate whether the computation of M on \sqcup is simulated by the computation of M on $code^i(M)$. As we are requiring candidate machines to be faithful, one first test we can apply is whether the termination behaviour of M on \sqcup and M on $code^i(M)$ 'match'. In other words, if M on \sqcup terminates, then so should M on $code^i(M)$; otherwise, M is clearly not universal. Similar comments apply if M on \sqcup does not terminate but M on $code^i(M)$ does. When both terminate, we may also conclude that M is not universal if the length of computation of M on \sqcup is longer than that of M on $code^i(M)$. As we are using process-oriented universality, for any universal machine M , the computation of M on $code^i(M)$ must be at least as long as that of M on \sqcup (and possibly longer).

Further analysis will presumably require a more direct comparison of the computation of M on \sqcup and M on $\text{code}^i(M)$, to check that each configuration of M on \sqcup appears in encoded form in the computation of M on $\text{code}^i(M)$.

8 Issues

A number of issues remain to be settled. We discuss a few of them here.

Is universality decidable?

This is not a question that appears to have been addressed previously, to the best of the author's knowledge. It would seem likely that the answer is no, i.e. that the universality of a given machine, for any universality context, is undecidable. Note that this is a more general problem than the one being addressed above, as this problem does not restrict the size of the machine M . In other words, if \mathcal{M} is the class of all Turing machines of any size (however Turing machines are defined), the problem is to determine whether an algorithm exists or not to determine whether any given $M \in \mathcal{M}$ is universal (however universality is defined).

Similar questions arise about the simulation of M on \sqcup by M on $\text{code}^k(M)$, i.e. whether or not it is possible to give an algorithm for any $M \in \mathcal{M}$ to determine whether the computation of M on \sqcup is simulated by M on $\text{code}^k(M)$. This problem has some minor variations, such as whether this holds for a given value of k , or whether it holds for some k , or for all k . It is well-known that it is undecidable whether M on \sqcup terminates or not (Sudkamp 2005). However, note that this problem involves a relative judgement, in that it is not whether the computation of M on \sqcup terminates or not, but whether this computation is simulated by M on $\text{code}^k(M)$. This means that the decidability of this problem is not obviously inconsistent with the undecidability of the termination of M on \sqcup . On the other hand, it seems intuitively likely that this problem is undecidable. Similar remarks apply to the more general question of whether it is decidable that the simulation of M_1 on w is simulated by M_2 on $\text{code}(M_1)w$.

Note that even if all of these problems are undecidable, this does not render our quest hopeless. As we are evaluating machines of a fixed size, there are only a finite number of machines under consideration at any particular time. This means that we are addressing a decidable instance of the more general problem. In particular, even if the general problem of testing for universality (i.e. over all possible Turing machines) is undecidable, testing for universality over a finite set of machines **is** decidable.

It should be noted that a property that we call the **finite decision anomaly** will apply here. This is that **any** decision problem over **any** finite set is decidable, as there are only a finite number of possible decisions – in particular, for a set of n elements and a decision with 2 outcomes, there are 2^n possible decisions, and for each one of these there is an algorithm that appropriately assigns ‘yes’ or ‘no’ for each element of the set. For the universality test, this means that for any class of machines M_n of size no more than n , there is *always* a decision procedure for universality.¹⁰ Hence even if the general test for universality is undecidable, then it is still possible to determine universality up to any given size of machine. This would be entirely analogous to the busy beaver problem, for which the general problem is undecidable, but for any given k , there is an algorithm to compute

the corresponding busy beaver value. Hence although there is no general algorithm, there is an algorithm for any given input value. This may be thought of as a concept of infinity which is particularly apt for theoretical computer science, in that for any finite value k we can compute the busy beaver value, but there is no single algorithm which will compute this value for an arbitrary n .

What proportion of machines are universal?

If we denote by M_n the set of all Turing machines of size $\leq n$, for a given universal context, we may define the universal proportion as the ratio

$$\frac{|U_{M_n}|}{|M_n|}$$

where U_{M_n} is the set of all universal machines in M_n . This leads us to ask a number of questions. For a given set of machines M_n , is there at least one universal context for which the universal proportion is non-zero? Is there more than one such universal context? What is the maximum universal proportion within a given universal context but across all applicable encoding functions? What is the maximum universal proportion across all universal contexts?

Criteria for universality

We have seen how Colmerauer's property can be used to derive a pseudo-universality test. Another possibility is to determine some other ‘test’ machine M and input w and to determine whether U on $\text{code}(M)w$ simulates M on w . It is tempting to choose M and w so that the computation of M on w is rather complex, on the grounds that if the corresponding computation of U on $\text{code}(M)w$ is not as complex, then U cannot be universal. However, this means that the evaluation of universality is potentially quite complicated. Hence finding such an M and w will require some care.

Busy beavers on universal machines

If we have a universal machine U , then U on $\text{code}(M)$ simulates M on \sqcup . Hence if M is a machine of size $\leq k$, then $\text{code}(M)$ is also of a limited size, and so we can evaluate the busy beaver value for M 's class of machines by evaluating a finite number of inputs to U . This suggests an alternative method for evaluating busy beaver machines may be to find an appropriate encoding and use it to find inputs for U . This is very much in the spirit of B  tfai (B  tfai 2009), who has shown how by ‘recombining’ machines and slightly modifying the class of machines used (such as allowing a transition to leave the tape pointer stationary, and not allowing for an explicit halting transition) it is possible to generate machines with greater productivities than the known busy beaver champions. This suggests that it may be interesting to evaluate busy beaver candidates by representing them as an appropriately size-restricted set of inputs to a universal machine (such as Colmerauer's machine). In particular, it seems natural to ask what is the smallest universal machine which is capable of simulating busy beaver machines. As a given size of machine will require only inputs of up to a given size (and hence a finite number overall), a ‘fully universal’ machine may not be required, but only a machine which can simulate machines of up to a given size on the blank input. Whether the minimal such machine is smaller than the smallest universal machine remains an open question.

9 Conclusions and Further Work

We have seen how the process of discovering universal machines, rather than constructing them, leads to a number of issues that need to be addressed before

¹⁰Although, somewhat paradoxically, knowing this gives us absolutely no information as to what the decision procedure may be.

an automated search can be carried out. We consider this the first step in a long process of experimentation to examine various classes of machines and apply various criteria for universality. Part of this process will be to examine various universality contexts and to investigate the differences between them, such as the proportion of universal machines found.

The question of the decidability of universality remains to be settled. Assuming this is answered in the negative, it then leads to the question of what criteria for pseudo-universality may be appropriate. We have discussed one such criterion, but there are presumably several others.

Another intriguing possibility is to consider two-dimensional Turing machines, i.e. machines which use a rectangular working space, rather the one-dimensional tape. In principle, such machines do not extend the capabilities of one-dimensional Turing machines. However, this does not preclude the possibility that universal machines may be smaller for such machines, or have some other differences that make them easier to discover.

References

- Norbert B{at}fai, Recombinations of Busy Beaver Machines, <http://arxiv.org/pdf/0908.4013>, September 7, 2009.
- George Boolos and Richard Jeffrey, *Computability and Logic*, 2nd edition, Cambridge University Press, 1980.
- Allen Brady, *The Determination of the value of Rado's noncomputable function $\Sigma(k)$ for four-state Turing machines*, Mathematics of Computation 40(162): 647-665, 1983.
- Cristian Calude, *Simplicity via Provability for Universal Prefix-free Turing Machines*, Theoretical Computer Science, in press.
- Alain Colmerauer, *On the Complexity of Universal Programs*, Proceedings of the Fourth International Conference on Machines, Computations and Universality 18-35, St. Petersburg, September 21-24, 2004. Published as Lecture Notes in Computer Science volume 3354, April, 2005.
- Alain Colmerauer, *Back to the Complexity of Universal Programs*, Proceedings of the Fourth International Conference on Constraint Programming, Sydney, September, 2008.
- Martin Davis, *A Note on Universal Turing Machines*, Automata Studies, Annals of Mathematics Studies 34:167-175, 1956.
- Martin Davis, *The Definition of Universal Turing Machines*, Proceedings of the American Mathematical Society 8(6):1125-1126, 1957.
- Patrick Fischer, *On Formalisms for Turing Machines*, Journal of the ACM 12: 570-581, October, 1965.
- Gabor Herman, *Simulation of One Abstract Computing Machine by Another*, Communications of the ACM 11(12):802 and 813, December, 1968.
- James Harland, *The Busy Beaver, the Placid Platypus, and other Crazy Creatures*, Proceedings of CATS'06 79-86, Hobart, January, 2006.
- James Harland, *An Inductive Theorem Prover for finding Busy Beaver Machines*, Proceedings of CATS'07 71-78, Ballarat, January, 2007.
- Alex Holkner, *Acceleration Techniques for Busy Beaver Candidates*, in Gad Abraham and Benjamin I.P. Rubenstein (eds.), *Proceedings of the Second Australian Undergraduate Students' Computing Conference* 75-80, December, 2004. ISBN 0-975-71730-8. Available from <http://www.cs.berkeley.edu/~benr/publications/auscc04>.
- Shen Lin and Tibor Rado, *Computer Studies of Turing Machine Problems*, Journal of the Association for Computing Machinery 12(2):196-212, 1964.
- Heiner Marxen and J{ur}gen Buntrock, *Attacking the Busy Beaver 5*, Bulletin of the EATCS 40:247-251, February 1990.
- Marvin Minsky, *Universality of $(p = 2)$ tag systems and a 4 symbol 7 state Turing machine*, AIM-33 AI Memo 33, Computer Science and Artificial Intelligence Laboratory, MIT, Cambridge, 1962.
- Edward Moore, *A Simplified Universal Turing Machine*, Proceedings of the ACM national meeting 50-54, Toronto, 1952.
- Turlough Neary, *Small universal Turing machines*, PhD Thesis, Department of Computer Science National University of Ireland, Maynooth, October 2008. Available from http://www.cs.may.ie/~tneary/tneary_Thesis.pdf.
- Turlough Neary and Damien Woods, *Four small universal Turing machines*, Proceedings of the 5th International Conference on Machines, Computations, and Universality 242-254, Orleans, 2007.
- Liudmila Pavlotskaya, *Dostatochnye usloviya razreshimosti problemy ustanovki dlja mashiny T'juring*, Avtomatu i Mashiny, 1978, 91-118 (Sufficient conditions for the halting problem decidability of Turing machines, in Russian).
- Lutz Priese, *Towards a Precise Characterization of the Complexity of Universal and Nonuniversal Turing Machines*, SIAM Journal of Computing 8(4):508-523, November, 1979.
- Tibor Rado, *On non-computable functions*, Bell System Technical Journal 41:877-884, 1963.
- Yurii Rogozhin, *Small universal Turing machines*, Theoretical Computer Science 168(2): 215-240, 1996.
- Claude Shannon, *A Universal Turing Machine with Two Internal States*, Annals of Mathematics 34:157-165, 1956.
- Thomas Sudkamp, *Languages and Machines: An Introduction to the Theory of Computer Science*, Addison-Wesley (3rd ed.), 2005.
- Alan Turing, *On Computable Numbers, with an Application to the Entscheidungsproblem*, Proceedings of the London Mathematical Society 2(42):230-265, 1937.
- Shigeru Watanabe, *4-symbol 5-state universal Turing machine*, Information Processing Society of Japan Magazine 13(9):588-592, 1972.
- Stephen Wolfram, *A New Kind of Science*, Wolfram Media, 2002.
- Damien Woods and Turlough Neary, *The complexity of small universal Turing Machines: A survey*, Theoretical Computer Science 410:443-450, 2009.